SST-Zesto: A Cycle Level Microarchitectural Simulator

1 Introduction

Zesto is a cycle-level simulator developed for modeling detailed x86 pipelines, including both out-of-order and in-order scheduling and execution that is designed to mirror current implementations, as well as many x86-specific microarchitecture features. The Zesto project was originated in Georgia Tech by Gabriel Loh and his students. More information on Zesto is available at http://zesto.cc.gatech.edu. SST-Zesto is a model of the core derived from the full Zesto model, bu leaving most of the core model intact while modifying interfaces to enable instantiation of multiple cores in a larger, SST many core simulation.

SST is the Structural Simulation Tookit developed in Sandia that enables simulation of diverse aspects of hardware and software in the co-design of extreme-scale architectures. The toolkit has a fully modular structure that enables extensive exploration of individual system parameters without the need for intrusive changes to the simulator. The MPI-based parallel simulation environment provides a high level of performance and the ability to model large systems. More information on SST is available at http://sst.sandia.gov.

This document describes necessary interfaces that should be established in order to build Zesto as a SST component and setup procedures to connect Zesto in a multi-node network with other uncore components.

2 SST-Zesto

This section describes how Zesto is migrated to SST by declaring new functionality and interfaces in the original Zesto core.

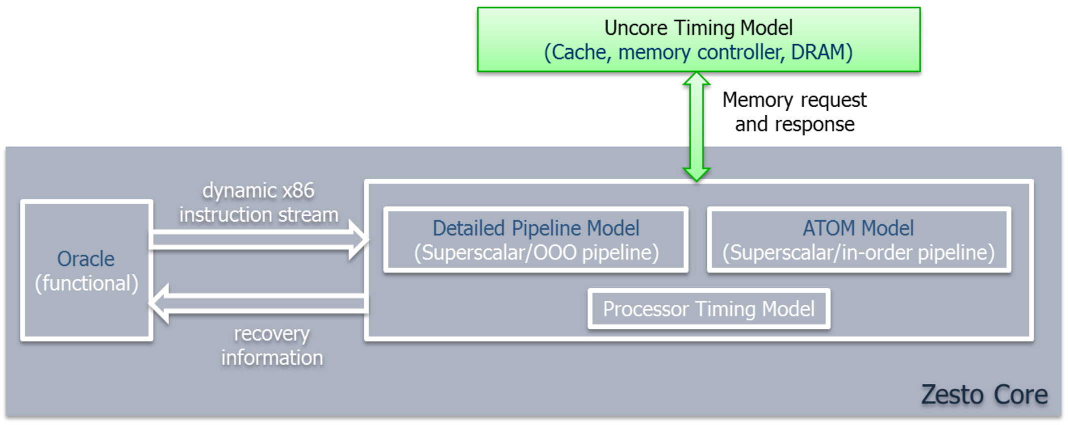
Uncore Timing Model
(Cache, memory controller, DRAM)
Memory request and response
dynamic x86 instruction stream
Oracle (functional)
recovery information
Detailed Pipeline Model (Superscalar/OOO pipeline)
ATOM Model (Superscalar/in-order pipeline)
Processor Timing Model
Zesto Core

Figure 1. Overall Zesto Structure

2.1 Overall Zesto Structure

The overall organization of the Zesto simulator includes a functional oracle simulator and a processor timing mode, as shown in Figure 1. The oracle simulator executes all instruction and provides the dynamic stream of instructions before the processor timing model takes all the stream information and performs detailed pipeline simulation and models timing. On a pipeline flush (e.g. due to branch misprediction or a load ordering violation), the processor timing model notifies the oracle of the relevant instruction for roll back, and the oracle uses this "undo" information to recover the previous state of the pipeline.

Currently Zesto supports two pipeline models/organizations/implementations: a Detailed Pipeline Model (DPM) which is based on modern Intel x86 superscalar out-of-order pipelines and an ATOM model which is based on x86 Atom in-order pipelines. Each pipeline model implements each of the five basic pipeline stages: Fetch, Decode, Allocation, Execute and Commit in a cycle-based manner.

All the memory requests generated by a Zesto core are handled by uncore timing models, such as cache, memory controller and DRAM. All these models can be defined to interface with a Zesto core when modeling many core systems.

Refer to available Zesto documents for more detailed description of Zesto internals.

2.2 Zesto migration to SST

The overall structure of SST-Zesto is shown in Figure 2.

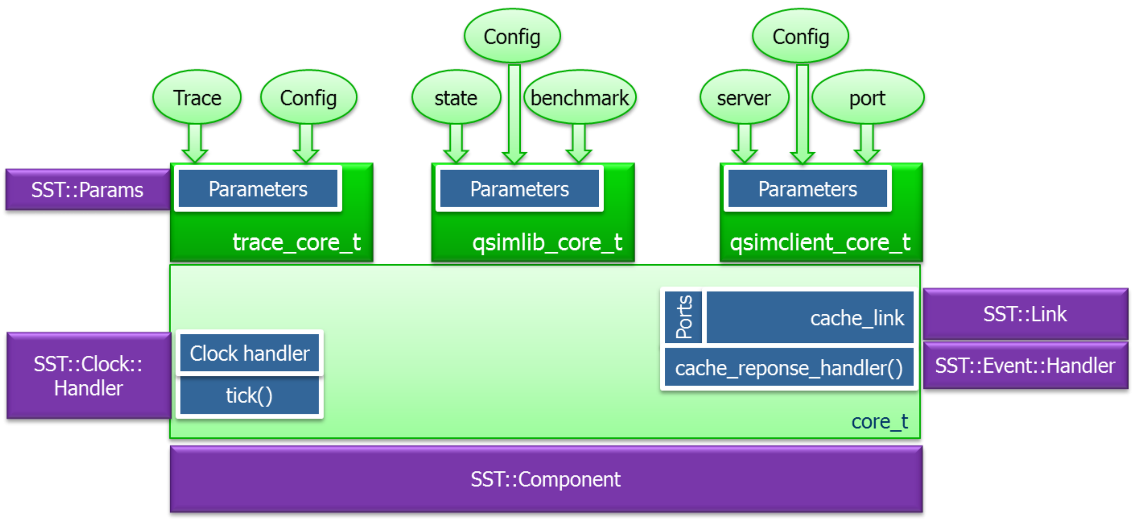

Figure 2. Zesto on SST

A Zesto core is defined as a component in SST using the SST Component class. All the functionalities defined in the original Zesto core, including oracle simulation and pipeline modeling, are wrapped in this component implementation.

Zesto is able to take multiple frontends as instruction inputs, including Traces generated by Qsim/PIN, Qsim library streams as well as Qsim server streams. Each frontend is defined as a derived class on top of a Zesto component to maintain a clear interface.

The trace frontend is supported with its corresponding class `trace_core_t`. The implementation of trace_core_t reads in the trace file specified in SST SDL configuration, and invokes the Zesto core to simulate the trace.

The class qsimlib_core_t defines the frontend taking Qsim library stream. Instead of reading trace, fetch stage in zesto core takes instructions generated by Qsim library to feed in the simulator. The Qsim library takes into Qsim state file and benchmark to generate OSDomain object which is shared by all qsimlib_core_t instances.

Qsim server stream is passed to zesto core through the frontend defined as qsimclient_core_t. Different from retrieving instructions from standalone Qsim library stream, all zesto cores in different network node can communicate with the same Qsim server to get instructions. Qsim server should be properly setup with Qsim state file and benchmark to generate its OSDomain object before SST-Zesto starts.

The cycle-by-cycle simulation of a Zesto core is performed by the clock handler tick() which is registered to SST in the class constructor. When this clock handler is invoked, the timing model of Zesto is used to advance the core state for one cycle.

Zesto can be connected to an uncore simple cache component through the link cache_link. The link is configured with cache_response_handler in the Zesto class constructor by calling the link configuration API defined in SST.

Both the Zesto configuration file and trace file will be part of SST params to zesto component, along with other parameters such as clock frequency and node id.

2.3 Simple Cache and Simple MC Component

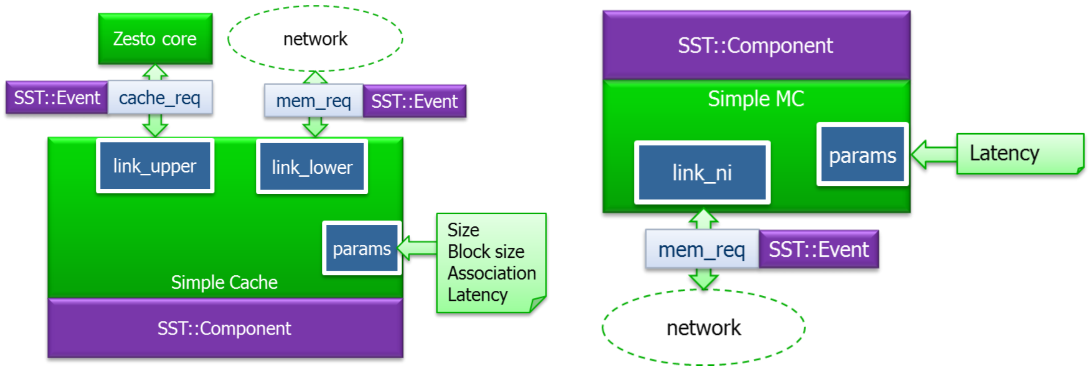

Figure 3. Simple Cache and Simple MC Component

Zesto is connected with a simple cache and a dummy memory controller to form a complete system, as shown in Figure 3.

Simple cache component is defined with two SST links: link_upper and link_lower. The link_upper link is connected to an upward terminal, which in this case is the Zesto core. The link_lower link is connected to next-level memory with/without the network. Cache configuration is specified through SST params, including size, block size, association and access latency. Inside the simple cache component, a hash table is maintained to simulate the cache hit/miss behavior.

The dummy memory controller is defined as Simple MC component with one link link_ni connected to either network or simple cache. Simple MC only takes latency as configuration parameter through SST params. Inside simple MC, the received memory requirement will be sent back to network after configured latency time.

The link message type passing between simple cache and Zesto core is defined as an SST event cache_req, while the messages routing around the network is defined as a SST event mem_req.

2.4 Iris Network Interface Controller

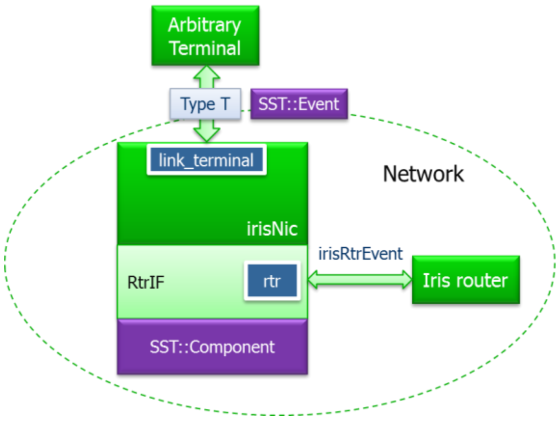

Figure 4. Iris Network Interface Controller

Both simple cache and simple MC can be connected to the Iris router through Iris Network Interface Contoller (NIC) component. Iris router is a developed SST component in SST repository. It is shipped with a general-purpose network interface RtrIF. However, in order to connect simple cache and simple MC to RtrIF, extra functionality is required, which is performed through IrisNIC. IrisNIC can be connected to an arbitrary terminal through its link link_terminal. The message passed on link_terminal can be an arbitrary type T inherited from SST Event. Event from terminal is first packed into Iris packets format in IrisNIC and then transmit to Iris router through packet sending APIs provided by RtrIF. Similarly, events from Iris router will be received and sent back to IrisNIC through RtrIF. IrisNIC unpacks and reorganizes the packets to send them back to the terminal.

More details of Iris Router and its packet format can be found in SST Iris documentations.

2.5 Typical Multi-Node Network Structure for Zesto

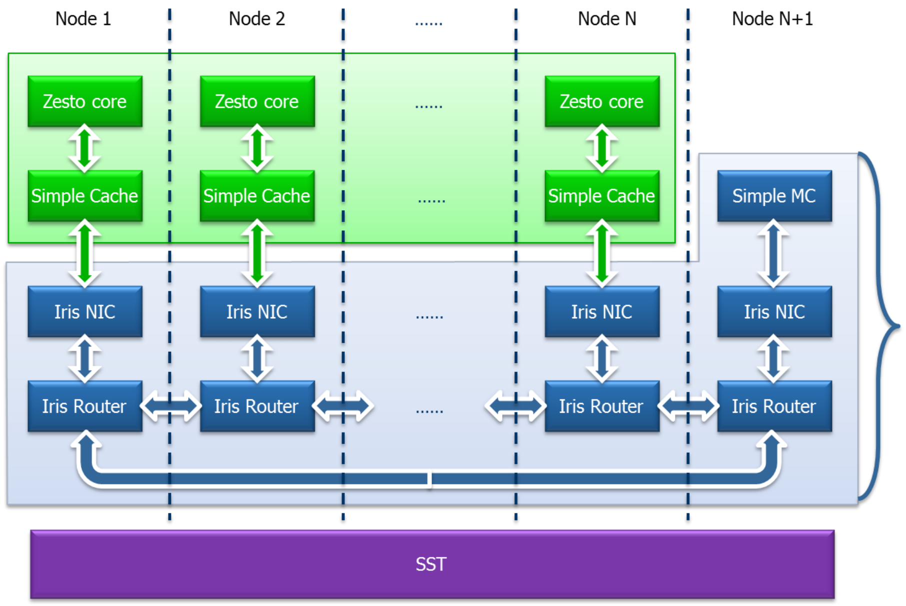

Figure 5. Typical Multi-Node Network

Figure 5 shows a typical N+1 Node network structure with the above SST components: Zesto, simple cache, simple MC, Iris NIC and Iris router. Each node contains either a pair of Zesto core and simple cache or a simple MC, which is connected to Iris network through Iris NIC. The Iris network can be configured with whatever topology supported by Iris router, e.g., mesh, ring, etc. The example in Figure 5 uses ring topology.

3 Running Zesto on SST

3.1 Zesto Directory Structure

Zesto is part of SST that can be found in sst-simulator/sst/elements/zesto. Table 1 gives brief description of each directory.

Directory	Description
zesto	Interface with SST.
zesto/core	Zesto core implementation.
zesto/core_config	Zesto core configuration files.
zesto/doc	Zesto documentation.
zesto/irisNIC	Iris network interface controller.
zesto/sdl	Sample SDL files.
zesto/simpleCache	Simple cache component.
zesto/simpleMC	Simple MC component.
zesto/trace	Sample trace file for zesto.

Table 1. Zesto Directory Structure

3.2 Building Zesto on SST

Zesto is built with SST. Refer to SST documentation for building procedure of SST.

Extra work should be done if users want to use Qsim frontend (both qsimlib and qsimclient). To set qsimlib_core_t/qsimclient_core_t as frontend, the users must install Qsim first. Qsim can be downloaded through https://sst-simulator.googlecode.com/svn/qsim/trunk. Follow the instruction in INSTALL file to install Qsim library, server, client and to generate state file. Qsim benchmarks are available at https://sst-simulator.googlecode.com/svn/qsim/benchmarks. Refer to README in the directory for detailed instructions of generating benchmark tar ball.

After that, users should modify the Makefile.am file under zesto directory. Put "-DUSE_QSIM" at the end of AM_CPPFLAGS which gives

```
AM_CPPFLAGS = \
    $(BOOST_CPPFLAGS) \
    $(MPI_CPPFLAGS)
```

And also put "-lqsim –lqsim-client" at the end of libzesto_la_LDFLAGS which gives

```
libzesto_la_LDFLAGS = -module -avoid-version –lqsim –lqsim-client
```

Reconfigure and rebuild SST according to SST building procedure.

3.3 Configure and Run Zesto

3.3.1 SDL configuration file

The system is setup by a SDL configuration file. General format of a SDL file can be found in the SST documentation. For Zesto systems, the SDL file should include all necessary configuration parameters for each component listed in Table 2.

Component	Parameter	Description
All Components	nodeId	Exclusive node id for each node in the network. All components in a node should have the same nodeId. One node includes a pair of Zesto and simple cache, or just a simple MC, as well as a Iris NIC and a Iris router (See Figure 5).
trace_core_t	clockFreq	Clock frequency of the core.
	configFile	Zesto configuration file. Refer to Zesto documentation for more details.
	traceFile	Traces generated by Qsim or Pin. Refer to Zesto documentation for format of trace file.
qsimlib_core_t	clockFreq	Clock frequency of the core.
	configFile	Zesto configuration file. Refer to Zesto documentation for more details.
	stateFile	State file required by Qsim library. Refer to Qsim documentation for generating state file.
	benchmark	Benchmark tar ball required by Qsim library. Refer to Qsim benchmark documentation for generating benchmark tar ball file.
qsimclient_core_t	clockFreq	Clock frequency of the core.
	configFile	Zesto configuration file. Refer to Zesto documentation for more details.
	server	Qsim server name. Can be the IP address or the network name of the server.
	port	Server port number.
simpleCache	size	Cache size.
	assoc	Association.
	block_size	Cache line size.
	hit_time	Cache hit access latency in ns.
	lookup_time	MSHR lookup latency in ns.
	mc_node_ids	Node Id of Simple MC.
simpleMC	clockFreq	Clock frequency of Simple MC.
	latency	Memory access latency.
irisNic	event_type	Event type name. Currently only “mem_req” is supported.
	num_vc	Virtual channel number. Always use 1.
	clock	Clock frequency of IrisNic.
	Node2RouterQSize_flits	Size of the router queue in Nic.
Iris router		Refer to Iris documentations.

Table 2. Parameters for Each SST Component

3.3.2 Qsim Setup

For `qsimlib_core_t` frontend specified in the SDL file, the parameter `stateFile` and `benchmark` must be set identical for all `qsimlib_core_t` instances. The behavior of the simulator is otherwise undefined. Note that stateFile should be generated in consistence with the number of cores specified in the SDL file. Refer to Qsim document for how to modify state file generating script to generate correct state file.

Qsim server should be started to support qsimclient_core_t frontend for zesto. This is done be executing qsim/remote/server with appropriate arguments for state file and benchmark. Now since all zesto cores fetch instructions from Qsim server as Qsim clients, there is no need for the users to specify state file and benchmark in SDL file. Instead, the instances are configured by server name and port which should be the same as the ones used by Qsim server setup.

3.3.3 Iris Network Setup

Zesto can be configured with or without Iris network. To configure Zesto without a network, the Zesto core is connected to simple cache which is then connected to simple MC directly. In this case, three components are used in one SDL file. The example SDL file for this configuration can be found in zesto/sdl/simple3.xml for trace frontend, zesto/sdl/simple3_qsimlib.xml for qsimlib frontend and zesto/sdl/simple3_qsimclient.xml for qsimclient frontend.

To configure Zesto with a network, a simple cache and simple MC must be connected to an Iris router through an Iris NIC. In this case, each node in the network may have either four components (zesto + simpleCache + IrisNic + Iris router) or three components (simpleMC + IrisNic + Iris router). Examples for this configuration for different frontends can be found in zesto/sdl/simple3_with_nic.xml, zesto/sdl/simple3_with_nic_qsimlib.xml and zesto/sdl/simple3_with_nic_qsimclient.xml.

Three example SDL files for a 5x1 network are given in zesto/sdl/conf5x1.xml, zesto/sdl/conf5x1_qsimlib.xml and zesto/sdl/conf5x1_qsimclient.xml for trace, qsimlib and qsimclient frontend respectively. The network structure is shown in Figure 5 with N=4.